Alberto Paoluzzi and Giorgio Scorzelli

BIM geometry with Julia Plasm

Functional language for CAD programming

May 27, 2024

Springer Nature

`1`	3
Pre/Post conds	`4` $\rightarrow$ `5`
Example	`6`

Contents

Chapter 3
Geometry and topology primer

It is implausible to set up a computer language and computational environment to define and deliver the geometric shape of construction elements and spaces without a pretty good understanding of basic geometric and topological concepts. Hence, this chapter introduces some basic notions about geometric spaces, operators, and properties, like linearity, that provide the computational foundation of shape definition and model construction. The essential elements of geometric spaces and cellular models, like affine transformations and simplicial, cubical, and polyhedral complexes, constitute the substrate of `Plasm` discourse for building hierarchical assemblies of AEC products of any scale and complexity.

3.1 Geometric Spaces

In mathematics and science, a space is a set of objects with some added structure. In this section we discuss the geometrical spaces used to represent inside a computer memory the actual objects of natural, artificial or virtual environments, or better their geometric models, in order to study, visualize and/or simulate some associated physical behavior. In particular, we deal in this section with linear, affine, and convex spaces as sets of vectors and/or points with selected properties.

3.1.1 Vector space

The concept of vector space is undoubtedly the most helpful instrument of mind invented by mathematicians for scientists, engineers, and architects, to represent and study formally or graphically both the primary and complex arrangements and behaviors of our natural and artificial environment.

Definition

A *vector space* (or *linear space*) $\mathcal{V}$ over a field $\mathcal{F}$ is a set, closed w.r.t. two composition rules (the output of composition belongs to the set of inputs).

The elements $\boldsymbol{v} \in \mathcal{V}$ are called *vectors*, and are often represented by an oriented arrow with given direction, orientation, and length. The elements $\alpha \in \mathcal{F}$ are called *scalars*. The sum of two non-zero vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{V}$ is a third vector $\boldsymbol{w} = \boldsymbol{u} + \boldsymbol{v} \in \mathcal{V}$ with direction and length different from both $\boldsymbol{u}$ and $\boldsymbol{v}$. We may write $\boldsymbol{v} + \boldsymbol{v} = 2\boldsymbol{v}$, and see here both the operations of a vector space: (a) addition of vectors, and (b) multiplication times a scalar.

The product of a vector by a scalar $\alpha\boldsymbol{v} = \boldsymbol{v}\alpha \in \mathcal{V}$ is a vector collinear with $\boldsymbol{v}$ and with different length if $\alpha \neq 1$. This explain the name "scalar" since a number "scales" (change the length of) the vector it multiplies.

The length of $\boldsymbol{u} = \alpha\boldsymbol{v}$ grows or shrinks w.r.t. $\boldsymbol{v}$ according to a positive $0 < \alpha < 1$. If $\alpha < 0$, then $\boldsymbol{u}$ has orientation opposite to $\boldsymbol{v}$.

Linear independence

A *linear combination* of vectors is a new vector that is defined as a sum of scalar multiples of other vectors. Let $\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_n \in \mathcal{V}$ and $\alpha_1, \alpha_2, ..., \alpha_n \in \mathcal{F}$, with $\mathcal{V}$ a vector space on the field $\mathcal{F}$ of scalar numbers. The vector

$$\boldsymbol{w} = \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \cdots + \alpha_n\boldsymbol{v}_n = \sum_{i=1}^{n} \alpha_i\boldsymbol{v}_i \in \mathcal{V}$$

is called a linear combination of vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_n$.

- Two or more vectors are said *linearly independent* if none of them can be written as a linear combination of the others;
- if at least one of them can be written as a linear combination of the others, then they are said *linearly dependent*.

Given a set of vectors, you can determine if they are linearly independent by writing the vectors as the columns of a matrix $\boldsymbol{A}$, and solving $\boldsymbol{A}\,\boldsymbol{x} = \mathbf{0}$.

If there are non-zero solutions, then the vectors are linearly dependent. If the only solution is $\boldsymbol{x} = \mathbf{0}$, then they are linearly independent.

The typical vector spaces in this book will be (a) the numeric field $\mathbb{R}$ for scalars and $\mathbb{R}^n$ for vector coordinates, with (b) $\dim = d$, for $0 \leq d \leq 3$.

Examples

Coding 3.1.1 (Matrices $m \times n$ are vector spaces) . Generate a random 100×100 matrix `A = rand(100,100)`. The default of `rand()` function are values of type `Float64` in `[0.0,1.0]`, so we ask the compiler to compute the matrix multiplication by the scalar π, denoted by the Greek symbol. The generated value is shown commented and simplyfied for printing in small space:

```
A = rand(100,100) * π   #=
100x100 Matrix{Float64}:
 0.901069  …  0.172854
 0.357548     0.8537
 ⋮         ⋱
 0.519136     0.857736      =#
```

Coding 3.1.2 (Random vector generation.) Then, we generate a 100-vector of random integers within the interval `[0,100]`, by using the Julia `0:100` iterator:

```
b = rand(0:100, 100)    #=
100-element Vector{Int64}:
 61
 85
  ⋮
 51          =#
```

Coding 3.1.3 (Multiplication matrix-vector.) This operation is implemented natively and very efficiently for big matrices in Julia, as well the product and the sum of dense matrices (the detail of printing depends on available space on output screen):

```
A * b        #=
100-element Vector{Any}:
 2786.383421064984
 2554.1378635659034
    ⋮
 2735.1205994270754 =#
```

Coding 3.1.4 (Addition matrix-vector.) Conversely, the sum of a matrix with a vector is not a linear operation, so we need to broadcast (. operator) the vector `b` on all columns of `A`:

```
A .+ b       #=
100x100 Matrix{Float64}:
 61.9011  …  61.1729
 85.3575     85.8537
  ⋮       ⋱
 51.5191     51.8577    =#
```

Summing up: while the `Matrix` by `Vector` multiplication is a native operation on the linear space of `Number`s, The addition of `Matrix` and `Vector` is not, and the broadcast operator must be used. Same for the sum of a matrix and a single scalar value. The reader should try.

Subspace

Let $\mathcal{V}$ be a vector space on the field $\mathcal{F}$. We say that $\mathcal{U} \subset \mathcal{V}$ is a *subspace* of $\mathcal{V}$ if $\mathcal{U}$ is a vector space with respect to the same operations. In particular, $\mathcal{U} \subset \mathcal{V}$ is a *subspace* of $\mathcal{V}$ if and only if:

1. $\mathcal{U} \neq \emptyset$;
2. for each $\alpha \in \mathcal{F}$ and $\boldsymbol{u}_1, \boldsymbol{u}_2 \in \mathcal{U}$, $\alpha\,\boldsymbol{u}_1 + \boldsymbol{u}_2 \in \mathcal{U}$

The *codimension* of a subspace $\mathcal{U} \subset \mathcal{V}$ is defined as $\dim \mathcal{V} - \dim \mathcal{U}$.

It may be useful to note that the intersection of subspaces is a subspace. In particular, if $\mathcal{U}_1, \mathcal{U}_2$ are subspaces of $\mathcal{V}$, then $\mathcal{U}_1 \cap \mathcal{U}_2$ is a subspace of $\mathcal{V}$.

Generators, Bases, and Coordinates

Span and generators

The smallest subset of vectors that can be generated by linear combinations of a subset $S \subset \mathcal{V}$ of linearly independent vectors is called *span* S.

The set S is therefore called a set of *generators*.

The *span* S is closed with respect to addition and multiplication, and hence is a *subspace* including the zero vector, which is contained in every subspace.

Bases and coordinates

The *basis* and *dimension* of the linear space $\mathcal{V}$ are a minimum set of generators for $\mathcal{V}$, and its number $d = \dim \mathcal{V}$ of elements, respectively.

Every basis of a linear space $\mathcal{V}$ has the same number d of elements.

When a basis for $\mathcal{V}$ has been fixed, i.e. an ordered minimal subset of generators $B \subset \mathcal{V}$ has been chosen, every vector $\boldsymbol{v} \in \mathcal{V}$ can be expressed *uniquely* as the linear combination of elements of B with scalars. The ordered tuple of such scalars is called the *coordinate* tuple, or the coordinates, of vector $\boldsymbol{v} \in \mathcal{V}$, and denoted as $[\boldsymbol{v}]$.

Remark 3.1 It is to mention that to represent vectors by coordinates requires that a minimum set of space generators and their ordering have been already chosen. We will say that the space has been p*arameterized*, since every vector is *uniquely* identified by the linear combination of the basis elements with a unique tuple of scalars.

Remark 3.2 The basis is often denoted by the ordered sequence $(\boldsymbol{e}_1, \dots, \boldsymbol{e}_d)$ of vector elements, or by the matrix $[\boldsymbol{e}_1 \cdots \boldsymbol{e}_d]$ of their coordinates by columns, where $\boldsymbol{e}_i = [0, 0, 1, \dots, 0]^t$ is a (column) tuple of zeroes, with only one element 1 in position i. The *standard basis* of a coordinate vector space is the set of vectors whose components are all zero, except one that equals 1.

Examples of vector spaces

The usual geometric example of vector space has the oriented arrows[1] as elements, summed with the parallelogram rule, and scaling related to elongation or shortening. Other examples are the linear spaces of matrices $\mathcal{M}_m^n(\mathbb{R})$ with real elements, m rows and n columns, and $\mathcal{M}_m(\mathbb{R})$ and $\mathcal{M}^n(\mathbb{R})$ of column and row vectors, respectively.

Linear space is also the space $\mathbb{P}_n(\mathbb{R})$ of real polynomial functions $p : \mathbb{R} \to \mathbb{R}$ such that $x \mapsto p^n(x)$ of degree $\le n$. In particular, $p^n(x) = a_0 + a_1\, x + a_2\, x^2 + \cdots + a_{n-1}\, x^{n-1} + a_n\, x^n$ is exactly a linear combination of a tuple of $n+1$ scalars a_k, $0 \le k \le n$, with the *power basis* of polynomials x^k, $0 \le k \le n$.

The Bernstein bases of polynomial space $\mathbb{P}_n(\mathbb{R})$

In geometric modeling and computer graphics, the Bernstein-Bézier basis of polynomials is fundamental. The $n+1$ Bernstein polynomials of degree n, defined as

$$B_k^n(x) = \binom{n}{k} x^k\, (1-x)^{n-k}, \qquad 0 \le k \le n$$

form a basis for the vector space $\mathbb{P}_n(\mathbb{R})$ of polynomials of degree at most n with real coefficients. Therefore, to implement the basis and to draw a graph of each basis function we have to consider: the degree `n` we are interested to; the ordinal number `k` of each function $(1 \le k \le n)$; and finally the independent variable x such that $x \mapsto p^n(x)$.

Script 3.1.1 (Pure functional style in Julia)
The julia function `B` given here returns the whole range of Bézier-Bernstein polynomials of any degree, and is very useful in curve geometric modeling.

```
B = n -> k -> u -> binomial(n,k) * u^k * (1-u)^(n-k)
```

Let us make some checks: `B(1)(0)(0.5) == B(1)(1)(0.5) == 0.5 #=> true`, for the degree-1 basis made by two polynomials `B(1)(0)` and `B(1)(1)`. The vhole basis of degree n is generated by the higher-level function

[1] Formally: the equivalence classes of equipollent oriented arrows. In Euclidean geometry, equipollence is a binary relation between directed line segments.

`Bernstein(n)` that we test in the quadratic case, where it returns a 3—element array of `Vector{Function}`:

Coding 3.1.5 (Generating Bernstein polynomial bases) The function `Bernstein(n)` is defined by mapping the partial function `B(n)` over the integer array `[0, 1, ..., n ]`.

```
Bernstein(n) = map(B(n), collect(0:n))
# => #7 Bernstein (generic function with 1 method)
Bernstein(2)
# => 3-element Vector{Function}
Bernstein(2)[2]
# => #9 (generic function with 1 method)
Bernstein(2)[2](0.1)
# => 0.18000000000000002
```

Now, we go to generate a discrete sequence of functions for the “vector” function `Bernstein(2)`, in order to test the implementation. Above we see also the second function of Bernstein basis of degree `n=2`, and finally its value `0.18` computed for `x = 0.1`.

Coding 3.1.6 (Sampling of third quadratic polynomial) The function `Bernstein(2)[3]` denotes the third function of the vector array `Bernstein (2)` of type `::Vector{Function}`, and we compute the three function values for `x=0.5`, that you can check in Figure 3.1.

```
Bernstein(2)[3] #=
37 (generic function with 1 method)    =#

Bernstein(2)[1](0.5) # => 0.25
Bernstein(2)[2](0.5) # => 0.5
Bernstein(2)[3](0.5) # => 0.25
```

Coding 3.1.7 (Sampling of third basis polynomial of degree 4) . We may look at the sequence of pairs $x, Bernstein(2)[3](x)$. A sampling of x values is created by `collect(0:0.1:1)`:

```
m = 10
for u in collect(0:1/m:1)
    println(" $(u), $(Bernstein(4)[3](u)) ")
end     #=
0.0, 0.0
0.01, 0.0005880600000000001
0.02, 0.00230496
0.03, 0.00508086
  ⋮     ⋮       =#
```

The Bernstein basis enjoy interesting properties. They are ≥ 0 for every value of the independent variable $x \in [0,1]$. Even more, for every x the elements of each basis sum to 1. Such bases are said *partition of unity.*

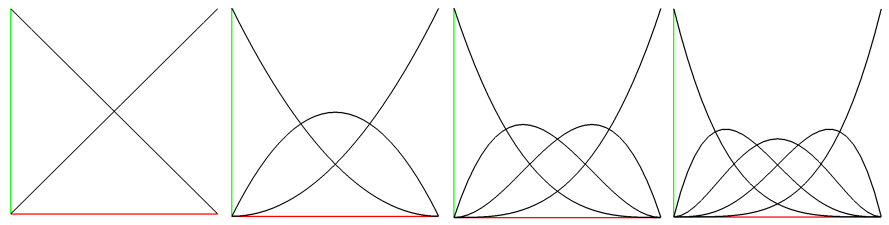

Fig. 3.1 The four images give the graph, in $[0,1] \times [0,1] \subset \mathbb{E}^2$, of Bernstein's basis of linear, quadratic, cubic, and quartic polynomials in linear spaces $\mathbb{P}_n(\mathbb{R})$, with $1 \leq n \leq 4$, and $\dim(\mathbb{P}_n(\mathbb{R})) = n+1 = 2$, 3, 4, and 5, respectively.

Coding 3.1.8 (Example of* `FL` *&* `Plasm` *Combinators) We remark on the incredible expressive power of `FL` programming inherited in `Plasm.jl`.

It is greatly useful to analyze goals and types of subexpressions, recuperating backward the development process to write the result expression. First, we define the 1D model `dom1D` of the function domain $[0,1]$ with 36 intervals, which is a geometric value of `Hpc` type:

```
dom1D = INTERVALS(1.0)(36)      #=
Hpc(MatrixNd([[1.0, 0.0], [0.0, 1.0]]), BuildMkPol[BuildMkPol(
   points=[[0.0], [0.027777777777777776], ... =#
```

Then, consider the $\mathbb{E} \to \mathbb{E}^2$ vector function `CONS(...)∘S1` of one variable, that is applied to the vertices of `dom1D::Hpc` by the `MAP` operator.

The selector function `S1` is used to extract the first (and unique) coordinate value from the internal data structure `point` array. The `CONS` operator, acting on argument of `Vector{Function}` type, transforms a function array into a vectorial function of coordinate functions `x(u)`, `y(u)`, etc.

Hence, we have `F = CONS(::Vector{Function})∘Sd` : $\mathbb{E}^d \to \mathbb{E}^n$, where `d` is the number of coordinates of domain vertices (one in this case), and `n` is the dimension of the embedding space, i.e., the number of coordinate functions inside the array argument, i.e. `CONS([,])`, two in this case:

```
F(k) = CONS([ID, Bernstein(4)[k]])∘S1       #=
#122 (generic function with 1 method)   =#
```

Finally, we note that expressions like `MAP(F)(dom1D)` are used to apply the `F` vector function to an object of `Hpc` type, normally to bend it. Remember that we generated an array of `Hpc` curves, for `k=1:n+1` — in our case with `n=4`. For this purpose, we need to finally apply the `STRUCT` operator, to transform the array of curves into a single `Hpc` value.

```
STRUCT( MAP(F(k))(dom1D) for k=1:n+1 )
```

Change of basis

It is often necessary, given a basis in a linear space, to compute a novel parameterization of the space with respect to a different, possibly more simple or useful basis. In other words, may be necessary to compute new coordinates for the vectors of the space, with respect to a new basis. This operation is called *change of basis*.

For concreteness, let we have vector data in a 3D linear space, where we need to compute their representation in a different basis. Let us denote the new basis as $(\boldsymbol{u}_i)$, and the standard one as $(\boldsymbol{e}_i)$, with $i \in \{1,2,3\}$. We may write the change of coordinates as a matrix map $\boldsymbol{T} : \mathcal{V} \to \mathcal{V}$, such that $[u_{ij}] \mapsto [e_{ij}]$, so transforming the old coordinates of the new basis into the standard basis.

Therefore we set $\boldsymbol{T}\,[u_{ij}] = [e_{ij}]$ and, since it is $[e_{ij}] = \mathbb{I}_3$, i.e., the 3×3 identity matrix, the solution is $\boldsymbol{T} = [u_{ij}]^{-1}$. Of course, this matrix exists provided that $\det\,[u_{ij}] \neq 0$. In other words, the vectors of the new basis must be linearly independent.

3.1.2 Affine space

In geometric modeling and computer graphics it is useful to distinguish between a space of vectors and a space of points (supported by vectors). A space of points, that provides for an operation of *displacement*, is called an *affine space*, and is represented here by the $\mathcal{A}$ symbol. In particular, we call *affine action* the function:

$$\mathcal{A} \times \mathcal{V} \to \mathcal{A},$$

so that the *displacement* from a point $\boldsymbol{a} \in \mathcal{A}$ to a point $\boldsymbol{b} \in \mathcal{A}$ is given. An affine space $\mathcal{A}$ is endowed with an operation of *difference* of points $\mathcal{A} \times \mathcal{A} \to \mathcal{V}$ where

$$\boldsymbol{a} + \boldsymbol{v} = \boldsymbol{b}, \quad \text{and hence} \quad \boldsymbol{b} - \boldsymbol{a} = \boldsymbol{v} \in \mathcal{V}$$

Remarks

We note that a **vector space** provides for an internal operation of (a) *sum* of two vectors and the external (b) *product* of a vector by a scalar, both returning a vector. Conversely, in an **affine space** we have an external operation of

(i) *difference* of points, returning a vector, and a (ii) sum of a point and a vector, called *displacement*, returning a point.

Even more, in a vector space $\mathcal{V}$ there is a distinguished element 0, the *zero vector*, which is contained in all subspaces. On the contrary, in an affine space $\mathcal{A}$ *all points are equivalent*, with no distinguished elements.

We usually say that a space of points has been parameterized when a *Cartesian system* (origin and basis) has been chosen. In order to use coordinates to make it easier to work with points we have to choose :

1. a point, called *origin*, to associate with the zero vector, and
2. an orthonormal *basis* of vectors.

The above steps consent to associate each point with the tuple of coordinates of its *displacement vector* from the origin.

In a vector space all the subspaces have at least a common element, the zero vector. Contrariwise, two affine subspaces may not have common elements. In such a case they are said *parallel*.

The dimension n of an affine space $\mathcal{A}$ is that of its supporting vector space $\mathcal{V}$. A common word for *affine subspaces* of d dimension is d-hyperplane, or d-hyperspace when $d > 3$. Lines and planes are affine subspaces of dimension 1 and 2, respectively.

Affine independence and local parameterization

In an affine space $\mathcal{A}$ of sufficiently high dimension n we say that two points are *affinely independent* when *non coincident*, three points when *non aligned*, four points when *non coplanar*, and so on.

In general, $d + 1$ points ($d \leq n$) are affinely independent when the d vectors defined by the differences $\boldsymbol{p}_k - \boldsymbol{p}_0$ of points from one of them are linearly independent.

The affine independence of a subset of $d+1$ points is often used to establish *local coordinate systems* on lines, planes and higher dimensional subsets of points. Let us choose two non coincident points $\boldsymbol{a}, \boldsymbol{b}$ on a 3D line. Any other point $\boldsymbol{p}$ of the line remains parameterized by the α scalar in the expression

$$\boldsymbol{p} = \boldsymbol{a} + \alpha(\boldsymbol{b} - \boldsymbol{a}).$$

Note that $\boldsymbol{b} - \boldsymbol{a}$ is a vector, $\alpha(\boldsymbol{b} - \boldsymbol{a})$ is a vector, $\boldsymbol{a} + \alpha(\boldsymbol{b} - \boldsymbol{a})$ is a point plus a vector, which is a point. The typing of our expression looks correct!

Analogously, let us choose three non colinear points $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ in 3D space. They are certainly non coincident, and of course fix a plane, i.e., an unique affine subspace of dimension 2 embedded in space. A parameterization of this plane is given by the pairs (α, β) of scalars in the expression

$$\boldsymbol{q} = \alpha(\boldsymbol{b} - \boldsymbol{a}) + \beta(\boldsymbol{c} - \boldsymbol{a}),$$

as the reader may immediately check. Similar local coordinates will hold in every affine d-subspace in n-dimensional space.

The typical affine space of points used in this book is the Euclidean space $\mathbb{E}^d$, $1 \leq d \leq 3$, usually furnished of a Cartesian system, with coordinates in `Float64`.

3.1.3 Convex space

Let us consider the Euclidean space $\mathbb{E}^d$, $d \in \{2, 3\}$ affine over the reals, that is the fundamental space of geometry, intended to represent physical space.

Affine subspaces become *convex sets* when a numeric constraint is imposed on the possible parameter values of an affine combination of points. Two points $\boldsymbol{a}, \boldsymbol{b} \in \mathcal{A}$ become the extreme elements of a *line segment* $\boldsymbol{p} = \alpha\, \boldsymbol{a} + (1-\alpha)\, \boldsymbol{b}$ as set of points, by adding the further constraint $\alpha + \beta = 1$ to the parameter values $\alpha, \beta \geq 0$, so posing $\beta = 1 - \alpha$. Analogously, setting $\alpha + \beta + \gamma = 1$ and $\alpha, \beta, \gamma \geq 0$ constraints the set of points combination of three non aligned points $(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$.

Positive, Affine and Convex Combination of Points

Let $\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_n$ be affinely independent points in $\mathbb{E}^d$, and $\alpha_1, \alpha_2, \ldots, \alpha_n$ be scalar in $\mathbb{R}$. Their combination $\alpha_1\boldsymbol{p}_1 + \alpha_2\boldsymbol{p}_2 + \cdots + \alpha_n\boldsymbol{p}_n$ is said *positive*, *affine*, and *convex*, respectively, when

1. $\alpha_1, \alpha_2, \ldots, \alpha_n \geq 0$ (positive combination)
2. $\alpha_1 + \alpha_2 + \cdots + \alpha_n = 1$ (affine combination)
3. $\alpha_1 + \alpha_2 + \cdots + \alpha_n = 1$ and $\alpha_1, \alpha_2, \ldots, \alpha_n \geq 0$ (convex combination)

It may be interesting to verify that the affine combination of points is a point. Let us eliminate the α_1 scalar using the unitary sum of scalars constraint:

$$\begin{aligned} \boldsymbol{p} &= (1 - \alpha_2 - \cdots - \alpha_n)\boldsymbol{p}_1 + \alpha_2\boldsymbol{p}_2 + \cdots + \alpha_n\boldsymbol{p}_n && (3.1) \\ &= \boldsymbol{p}_1 + \alpha_2(\boldsymbol{p}_2 - \boldsymbol{p}_1) + \cdots + \alpha_n(\boldsymbol{p}_n - \boldsymbol{p}_1) && (3.2) \end{aligned}$$

which, of course, is a point in $\mathbb{E}^d$.

A convex combination is positive and affine.

The set of all convex combinations of points $C \subset \mathbb{E}^n$ is the *convex hull* of C. The convex hull is the smallest compact set containing the points in C. It is the intersection of all compact sets of $\mathbb{E}^d$ which contain $\operatorname{hull} C$.

Convex coordinates of a point $\boldsymbol{c} \in \operatorname{hull} C$ are the scalars whose convex combination with C elements produces $\boldsymbol{c}$. If $C \subset E^n$ has $n+1$ affinely independent elements, i.e., it is a simplex (see Section **??**) the convex coordinates of each $\boldsymbol{c} \in \text{hull } C$ are *unique*.

Characterization of affine pyramids

Speaking of local parameterization of affine subspaces, we did not fix any constraint on the signs of the scalar parameters (usually in the field $\mathbb{R}$ of reals). If all the scalars are positive, then all the spanned points stay inside the interior of a planar (or solid) angle contained within the lower-dimensional affine subspaces generated. It would be not difficult to see that the whole subspace will be partitioned by an arrangement of subspaces centered on the fixed point of the set of linearly independent vectors. As an example, consider the partitioning of 3D space generated by the $8 = 2^3$ octants defined by 4 non coplanar points, and in particular those generated when three points are at unit distance by the first, and the three difference vectors are pairwise orthogonal.

Positive, Affine and Convex coordinates

Affine Coordinates for Convex Sets
  Barycentric Coordinates for Convex Sets
  Test for boundary points: interior and exterior

3.2 Cellular models

This section discusses computational topology concepts used to compute the 2D/3D space partition induced by a collection of geometric objects of dimension 1D/2D, respectively. Cellular models are are often called *meshes* in engineering and design jargon. A mesh is a representation of a domain by discrete cells. In many areas of geometric/numeric computational engineering, including geo-mapping, computer vision and graphics, finite element analysis, medical imaging, geometric design, and solid modeling, one has to compute incidences, adjacencies and ordering of mesh cells, generally using disparate and incompatible data structures and algorithms. Only sparse vectors and matrices are used in `Plasm` to compute the linear spaces of chains, called chain complex, from dimension zero to three.

Topological definitions

A *complex* is a graded set $S = \{S_i\}_{i \in I}$ *i.e.* a family of sets, indexed here over $I = \{0, 1, 2, 3\}$. We use two different but intertwined types of complexes, and specifically complexes of *cells* and complexes of *chains*. Their definitions and some related concepts are given in this section. Greek letters are used for the cells of a space partition, and roman letters for chains of cells, coded as either (un)signed integers or sparse arrays of (un)signed integers.

Definition 3.1 (d-Manifold) A *manifold* is a topological space that resembles a flat space locally, *i.e.*, near every point. Each point of a d-dimensional manifold has a neighborhood that is homeomorphic[2] to $\mathbb{E}^d$, the Euclidean space of dimension d. Hence, this geometric object is often referred to as d-manifold.

Homeomorphic neighborhood means "topologically equivalent", like a rubber patch, that can be stretched without changing its topology.

Definition 3.2 (Cell) A *p-cell* σ is a p-manifold with boundary $(0 \leq p \leq d)$ which is piecewise-linear, connected, possibly non convex, and not necessarily contractible. This definition refers to cellular complexes used in this paper and is different from other ones because a cell is neither simplicial, nor convex, nor contractible.

In `Plasm`, cells may contain internal holes; cells of CW-complexes [13] are, conversely, contractible to a point. We deal with *Piecewise-Linear* (PL) cells of dimensions 0, 1, 2, and 3, respectively. It should be noted that 2- and 3-cells may contain holes, while remaining connected. In other words, `Plasm` cells are p-polyhedra, *i.e.* segments, polygons and polyhedrons embedded in 2D or 3D space.

Definition 3.3 (Cellular complex) A *cellular p-complex* is a finite set of cells that have at most dimension p, together with all their r-dimensional boundary faces $(0 \leq r \leq p)$. A *face* is an element of the PL boundary of a cell, that satisfy the *boundary compatibility* condition. Two p-cells α, β are said boundary-compatible when their point-set intersection contains the same r-faces $(0 \leq r \leq p)$ for both α and β.

A cellular p-complex is said *homogeneous* when each r-cell $(0 \leq r \leq p)$ is face of a p-cell.

Definition 3.4 (Skeleton) The s-skeleton of a p-complex Λ_p $(s \leq p)$ is the set $\Lambda_s \subseteq \Lambda_p$ of all r-cells $(r \leq s)$ of Λ_p. Every skeleton of a regular complex is a regular subcomplex. The difference $\Lambda_r - \Lambda_{r-1}$ of two skeletons is the set U_r of r-cells.

[2] Homeomorphic means opologically equivalent.

Definition 3.5 (Support space) The support space Λ of a cellular complex is the point-set union of its cells.

Definition 3.6 (Characteristic function) Given a subset S of a larger set A, the characteristic function $\chi_A(S)$, also called the *indicator function*, is the function defined to be identically one on S, and zero elsewhere. [34].

3.2.1 Simplicial complex

Definition 3.7 (Join operation.) The *join* of two compact sets of points $P, Q \subset \mathbb{E}^n$ is the set PQ of convex combinations of points in P and in Q. The join operation is associative and commutative.

Definition 3.8 (Simplex.) A *d-simplex* $\sigma_d \subset \mathbb{E}^n$ $(0 \leq d \leq n)$ may be defined as the repeated join of $d+1$ affinely independent points, called *vertices*.

A d-simplex can be seen as a d-dimensional triangle: a 0-simplex is a *point*, a 1-simplex is a *segment*, a 2-simplex is a *triangle*, a 3-simplex is a *tetrahedron*, and so on.

Coding 3.2.1 (Plasm d-dimensional simplex) A d-simplex is generated by the `Plasm` package by the `SIMPLEX` multi-dimensional function:

```
julia> using Plasm

julia> SIMPLEX::Function
SIMPLEX (generic function with 1 method)
```

Coding 3.2.2 (Dataset associated to simplices.) The simplex datasets follow, for the first integer parameters. Note that `SIMPLEX(d)` contain $d+1$ coordinate points of dimension d, and that `hulls` fields contain $d+1$ indices of points, i.e. a single *convex* cell:

```
SIMPLEX(0)          #=
Hpc(MatrixNd(1), Geometry([Float64[]], hulls=[[1]])) =#

SIMPLEX(1)          #=
Hpc(MatrixNd(2), Geometry([[0.0], [1.0]], hulls=[[1, 2]])) =#

SIMPLEX(2)          #=
Hpc(MatrixNd(3), Geometry([[0.0, 0.0], [1.0, 0.0], [0.0, 1.0]],
    hulls=[[1, 2, 3]])) =#

SIMPLEX(3)          #=
Hpc(MatrixNd(4), Geometry([[0.0, 0.0, 0.0], [1.0, 0.0, 0.0],
    [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], hulls=[[1, 2, 3, 4]])) =#
```

```
SIMPLEX(4)          #=
Hpc(MatrixNd(5), Geometry([[0.0, 0.0, 0.0, 0.0], [1.0, 0.0, 0.0,
     0.0], [0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], [0.0,
    0.0, 0.0, 1.0]], hulls=[[1, 2, 3, 4, 5]])) =#
```

For the sake of visual simplicity, we remove the `julia>` prompt for Plasm scripts, and use the multilinear comment to reduce their visual rumor.

Definition 3.9 *Skeleton and Faces.* The set $\{\boldsymbol{v}_0, \boldsymbol{v}_1, \dots, \boldsymbol{v}_d\}$ of vertices of σ_d is called the *0-skeleton* of σ_d. The s-simplex generated from *any* subset of $s+1$ vertices $(0 \leq s \leq n)$ of σ_d is called an *s-face* of σ_d.

Remark 3.3 Let us notice, from the definition, that a simplex may be considered both as a purely *combinatorial object* and as a *geometric object*, i.e. as the compact point-set defined by the convex hull of a discrete set of points.

Definition 3.10 (Simplicial Complex) A set Σ of simplices is called a *triangulation*. A *simplicial complex*, often simply denoted as *complex*, is a triangulation Σ that verifies the following conditions:

1. if $\sigma \in \Sigma$, then any face of σ belongs to Σ;
2. if $\sigma, \tau \in \Sigma$, then either $\sigma \cap \tau = \emptyset$, or $\sigma \cap \tau$ is a face of both σ and τ.

A simplicial complex can be considered a *well-formed* triangulation. Such kind of triangulations are widely used in engineering analysis, e.g., in topography or in finite element methods.

Coding 3.2.3 (binomial numbers and simplex faces) There are

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

k-faces in a n-simplex. Testing this statement is simple and very elegant with Julia. We may verify, e.g., that `simplex(5)` faces are a set of 2^5 elements:

```
julia> [binomial(5,k) for k=0:5]'
1×6 adjoint(::Vector{Int64}) with eltype Int64:
 1  5  10  10  5  1

julia> [binomial(5,k) for k=0:5] |> sum == 2^5
true
```

Coding 3.2.4 A simple coding example produces the simplicial complex providing the triangulation of the boundary of a convex hull.

```
julia> p = rand(6,3)                    #=
6×3 Matrix{Float64}:
 0.456121  0.340689  0.523394
 0.670731  0.920846  0.810581
 0.511325  0.83709   0.765548
 0.27295   0.344676  0.246891
 0.155611  0.262588  0.372059
 0.997037  0.689132  0.594624          =#
```

The `p` matrix holds 6 random `3D` points. The `q` matrix receives an array of arrays:

```
julia> q = [p[k,:] for k=1:size(p,1)]              #=
6-element Vector{Vector{Float64}}:
 [0.4561213293752391, 0.34068894716553066, 0.5233939444809501]
 [0.6707310911896536, 0.9208461259235498, 0.8105811405026019]
 [0.5113250218604997, 0.8370897242704682, 0.7655476328700838]
 [0.2729499977250678, 0.3446764528053594, 0.24689130455109154]
 [0.15561059083290985, 0.26258788197490757, 0.37205895443742]
 [0.9970371401112009, 0.6891321979374976, 0.5946242627157257]
    =#
```

Their convex hull is computed by the `Plasm` function `CONVEXHULL` and stored as `Hpc` data structure:

```
julia> CONVEXHULL(q)                    #=
Hpc(MatrixNd(4), Geometry([[0.4561213293752391,
    0.34068894716553066, 0.5233939444809501],
    [0.6707310911896536, 0.9208461259235498,
    0.8105811405026019], [0.5113250218604997,
    0.8370897242704682, 0.7655476328700838],
    [0.2729499977250678, 0.3446764528053594,
    0.24689130455109154], [0.15561059083290985,
    0.26258788197490757, 0.37205895443742], [0.9970371401112009,
     0.6891321979374976, 0.5946242627157257]], hulls=[[1, 2, 3,
    4, 5, 6]]))               =#
```

And finally the dataset is transformed into a `Lar` data structure, which contains all the boundary polygons `:FV` and edges `:EV`.

```
julia> LAR(CONVEXHULL(q))                   #=
Lar(3, 3, 6, [0.27294999772507 0.67073109118965 …
    0.15561059083291 0.5113250218605; 0.34467645280536
    0.92084612592355 … 0.26258788197491 0.83708972427047;
    0.24689130455109 0.8105811405026 … 0.37205895443742
    0.76554763287008], Dict{Symbol, AbstractArray}(:CV => [[1,
    2, 3, 4, 5, 6]], :FV => [[1, 2, 3], [2, 3, 4], [1, 4, 5],
    [1, 3, 4], [1, 5, 6], [1, 2, 6], [4, 5, 6], [2, 4, 6]], :EV
    => [[2, 3], [1, 3], [1, 2], [3, 4], [2, 4], [1, 4], [4, 5],
    [1, 5], [5, 6], [1, 6], [2, 6], [4, 6]]))               =#
```

We remark that the `Hpc` describes only the *convex cells* of its dataset. Conversely, the `Lar` data structure gives explicitly all cells of any dimension. □

The *order* of a complex is the maximum order of its simplices. A complex Σ_d of order d is also called a *d-complex.* A d-complex is said to be *regular* or *pure* if each simplex is a face of a d-simplex. A regular d-complex is homogeneously d-dimensional.

The *combinatorial boundary* $\Sigma_{d-1} = \partial\sigma_d$ of a simplex σ_d is a simplicial complex consisting of all proper s-faces ($s < d$) of σ_d.

Two simplices σ and τ in a complex Σ are called *s-adjacent* if they have a common s-face. Hereafter, when we refer to adjacencies into a d-complex, we intend to refer to the maximum order adjacencies, i.e. to $(d-1)$-adjacencies. $\mathcal{K}_s$ ($s \leq d$) denotes the set of s-faces of Σ_d.

Simplex orientation

The ordering of 0-skeleton of a simplex implies an *orientation* of it.

Definition 3.11 (Ordering of simplex vertices) The simplex can be oriented according to the even or odd permutation class of its 0-skeleton.

The two opposite orientation of a simplex will be denoted as $+\sigma$ and $-\sigma$.

Definition 3.12 (Coherent orientation) Two simplices are *coherently oriented* when their common faces have opposite orientation. A complex is *orientable* when all its simplices can be coherently oriented.

It is assumed that:

1. the two orientations of a simplex represent its relative interior and exterior;
2. the two orientations of an orientable simplicial complex analogously represent the relative interior and exterior of the complex, respectively;
3. the boundary of a complex maintains the same orientation of the complex.

The volume associated with an orientation of a simplex (or complex) is positive, while the one associated with the opposite orientation has the same absolute value and opposite sign. It is assumed that the bounded object has positive volume. It is also assumed that either a minus sign or a multiplying factor -1 denote a complementation, i.e. an opposite orientation of the simplex, which can be explicitly obtained by swapping two vertices in its ordered 0-skeleton. For example:

$$+\sigma_3 = \langle \boldsymbol{v}_0, \boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3 \rangle, \qquad -\sigma_3 = \langle \boldsymbol{v}_1, \boldsymbol{v}_0, \boldsymbol{v}_2, \boldsymbol{v}_3 \rangle.$$

Definition 3.13 (Volume of a 3-simplex) The volume of a 3-simplex is a 2-cochain (see Section 3.3) from the cycle of its 2-faces (boundary triangles) to $\mathbb{R}$. By definition, a 2-cochain is a map from 2-chains to $\mathbb{R}$.

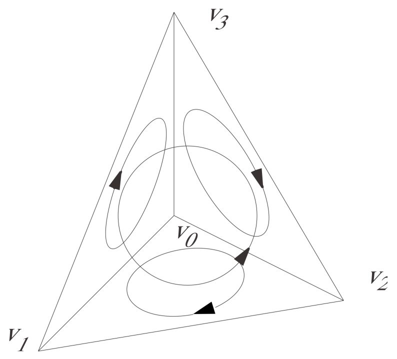

Fig. 3.2 Coherent orientation of the 2-faces of a 3-simplex

Facet extraction

The $(d-1)$-faces of a d-dimensional simplex or complex are also called *facets* since no ambiguity may arise. It is possible to see [39] what follows:

Theorem 3.1 (Whitney, 1957) *The oriented facets $\sigma_{d-1}^{(i)}$ $(0 \leq i \leq d)$ of the oriented d-simplex $\sigma_d = +\langle \boldsymbol{v}_0, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_d \rangle$ are obtained by removing the i-th vertex $\boldsymbol{v}_i$ from the 0-skeleton of σ_d:*

$$\sigma_{d-1}^{(i)} = (-1)^i (\sigma_d - \langle \boldsymbol{v}_i \rangle), \qquad 0 \leq i \leq d. \tag{3.3}$$

The 0-skeleton of $\sigma_{d-1,(i)}$ is therefore obtained by removing the i-th vertex from the 0-skeleton of σ_d and either by swapping a pair of vertices or, better, by inverting the simplex sign, when i is odd.

Coding 3.2.5 (Facets of a d-simplex.) A julia function is given here to compute combinatorially all the oriented facets of a `simplices` arrays of d-simplexes. A precondition is that all `Vector{Int64}` in the input sequence have the same length.

```julia
function simplexfacets(simplices)
   @assert hcat(simplices...) isa Matrix
   out = Array{Int64,1}[]
    for simplex in simplices
        for v in simplex
            facet = setdiff(simplex,v)
            push!(out, facet)
        end
    end
    # remove duplicate facets
    return sort(collect(Set(out)))
end
```

If the `expr` after `@assert` is `false`, the program stops in `AssertionError: expr` □

Coding 3.2.6 (Execution example.) We can start from an array of simplices of the same length and iterate on their facets, to get a whole simplicial complex.

```
FV=[[123],[124],[134],[234]] #=
[[123],[124],[134],[234]] =#
EV = simplexfacets(FV)  #=
[[12],[13],[14],[23],[24],[34]] =#
VV = simplexFacets(EV)  #=
[[1], [2], [3], [4]]     =#
simplex(3,complex=true).C   #=
Dict(:c1v => EV, :c3v => [[1,2,3,4], :c0v => VV], :c2v => FV) =#
```

We can also generate directly the whole bounday complex of a multidimensional simplex

Coding 3.2.7 (Generation of boundary complex) We can olso directly generate the `Lar` boundary complex of a simplex of any degree:

```
simplex(4,complex=true).C           #=
Dict{Symbol, AbstractArray} with 5 entries:
  :C4V => [[1, 2, 3, 4, 5]]
  :C3V => [[1, 2, 3, 4], [1, 2, 3, 5], [1, 2, 4, 5], [1, 3, 4,
    5], [2, 3, 4, 5]]
  :C0V => [[1], [2], [3], [4], [5]]
  :C2V => [[1, 2, 3], [1, 2, 4], [1, 2, 5], [1, 3, 4], [1, 3,
    5], [1, 4, 5], [2…
  :C1V => [[1, 2], [1, 3], [1, 4], [1, 5], [2, 3], [2, 4], [2,
    5], [3, 4], [3, …  =#
simplex(4,complex=true).V           #=
4×5 Matrix{Float64}:
 0.0  1.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  0.0  1.0              =#
```

Coding 3.2.8 (Simplex generation in `Hpc`) Convnersely, the `Plasm` data structure `Hpc` contains a single convex cell:

```
SIMPLEX(4)          #=
Hpc(MatrixNd(5), Geometry([[0.0, 0.0, 0.0, 0.0], [1.0, 0.0, 0.0,
     0.0], [0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], [0.0,
    0.0, 0.0, 1.0]], hulls=[[1, 2, 3, 4, 5]]))          =#
```

Definition 3.14 (Simplicial extrusion)
The prism over a simplex $\sigma_d = \langle \boldsymbol{v}_0, \ldots, \boldsymbol{v}_d \rangle$, defined as the set $P_{d+1} := \sigma_d \times [a, b]$, with $[a, b] \subset \mathbb{E}$, will be called *simplicial* $(d+1)$*-prism.* An oriented complex which triangulates P_{d+1} can be defined combinatorially, by using a closed form formula for its $\mathcal{K}_{d+1}$ skeleton:

$$\mathcal{K}_{d+1} = \{\sigma_{d+1,(i)} = (-1)^{id} \langle \boldsymbol{v}_i^a, \boldsymbol{v}_{i+1}^a, \ldots, \boldsymbol{v}_d^a, \boldsymbol{v}_0^b, \boldsymbol{v}_1^b, \ldots, \boldsymbol{v}_i^b \rangle, \quad 0 \leq i \leq d\}$$

where $\boldsymbol{v}_i^a = (\boldsymbol{v}_i, a)$ and $\boldsymbol{v}_i^b = (\boldsymbol{v}_i, b)$.

Closed formulas to triangulate the $(d+1)$-prism over a d-complex in a time linear with the size of the output, while computing also the d-adjacencies between the resulting $(d+1)$-simplices, can be found in [12, 11].

Simplicial grids

In layout design, a grid system provides a framework of intersecting vertical and horizontal lines. Designers use this framework to place and align text, images, and other elements.

Analogously, a *simplicial grid*, is a geometric grid structure made by simplicial cells of same dimension aligned on a layout grid 1D, 2D, 3D, etc. They are mainly used for domain decomposition, where to map coordinate functions in order to create curved structures, i.e., proper curves, curved surfaces, and curved solids.

The whole curved complex is easily generated by mapping the vertices of a `Lar` grid.

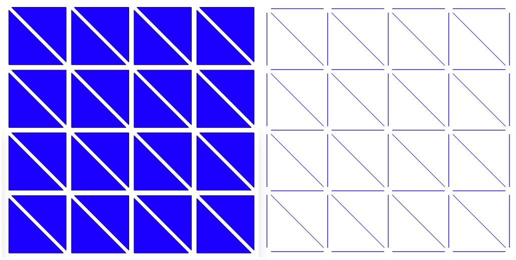

Fig. 3.3 The simplicial 2- and 1-complex (shown here slightly exploded) are generated by *extrudecomplex()* using the same pattern, both starting from *model1D*. The 1-complex is derived by the 2-complex using *simplexfacets()*.

Coding 3.2.9 (1D Extrusion)

```
V1 = [ 0.0 1.0 2.0 3.0 4.0 5.0 ]
EV = [[1, 2], [2, 3], [3, 4], [4, 5]]
```

```
mod1D = Lar(V1, Dict(:c1v => EV))
V2 = [mod1D.V; zeros(size(mod1D.V,2))']
GL.VIEW(GL.GLExplode(V2, map(x->[x],EV),1.1,1.1,1.1,2,1));
```

The mod2D and mod3D grids are generated by the following code.

Coding 3.2.10 (2D Extrusion)

```
mod2D = extrudecomplex(mod1D::Lar, [1, 1, 1, 1])
FV = collect(values(mod2D.C))[1]
EV = simplexfacets(FV)
GL.VIEW(GL.GLExplode(mod2D.V, map(x->[x],EV),1.2,1.2,1.2,4,1));
```

Coding 3.2.11 (3D Extrusion)

```
mod3D = extrudecomplex(mod2D::Lar, [1,-2,1])
CV = collect(values(mod3D.C))[1]
FV = simplexfacets(CV)
EV = simplexfacets(FV)
GL.VIEW(GL.GLExplode(mod3D.V,map(x->[x],EV),2,2,2,99,1));
```

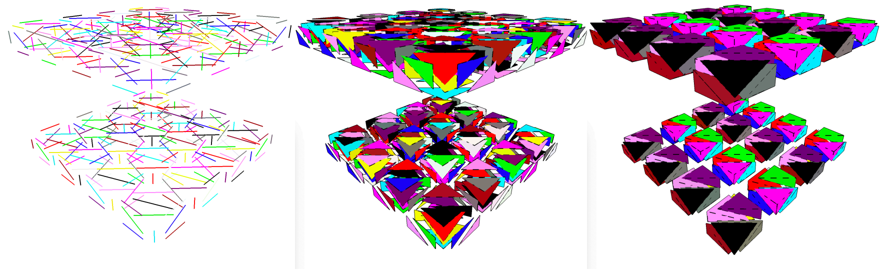

Fig. 3.4 Exploded simplicial grids of dimensions 1 (lines), 2 (surfaces), and 3 (solids). The simplicial 2-complex is generated by *simplexfacets()* from the 3-complex *extrudecomplex(simplex2D)*, with *simplex2D* in turn starting from *model1D*, i.e., from the expression generating the simplicial 1-complex .

3.2.2 Cubical complex and grid

We introduce now the definition and use of multidimensional *grids* of *cuboidal*, and the more general *Cartesian product* of cellular complexes. Such operators, depending on the dimension of their input, may generate either *full-dimensional* (i.e. solid) output complexes, or *lower-dimensional* complexes of dimension d embedded in Euclidean n-space, with $d \leq n$.

Regular grid

Regular cubical grids appear in finite element analysis, finite volume methods, finite difference methods, and in general for discretization of parameter spaces.

Definition 3.15 (Regular cuboidal grid) A regular grid is a tessellation of n-dimensional Euclidean space by congruent parallelotopes (e.g. bricks). Its opposite is irregular grid. We prefer to call these objects "cuboidal" complexes, in order to remark their multidimensional cheracter in `Plasm`.

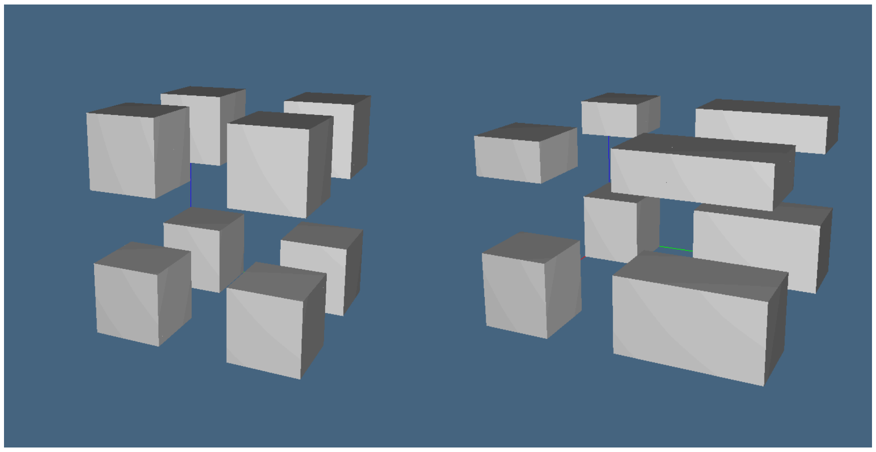

Fig. 3.5

E.g., just think to a mesh of 3D cubes in three-dimensional space for the first case, and to the (non-manifold) framework of skeletal polygons of such cubic cells for the second case. In particular, both n-dimensional *solid grids* of (hyper)-cuboidal cells and their d-dimensional *skeletons* ($0 \leq d \leq n$), embedded in $\mathbb{E}^n$, are generated by assembling the cells produced by a product of n either 0D or 1D cellular complexes, that in such lowest dimensions coincide with simplicial complexes.

In `Plasm` the cuboidal grids are not mandatory regular, and can be even unconnected.

Coding 3.2.12 (3D Cell Complex) Two simple examples are given here, and shown in Figure 3.5, of 3D regular and irregular cuboidal grids, respectively. Each `GRID` instance generates a 1D cellular complex, and two Cartesian product, denoted in `Plasm` by the `*` infix symbol, produces the 3D cell complex.

```
VIEW(GRID([1,-1,1]) * GRID([1,-1,1]) * GRID([1,-1,1]))
VIEW(GRID([1,-2,1]) * GRID([1,-1,2]) * GRID([1,-1,0.5]))
```

It is easy to imagine that the negative parameters denote empty space, while the positive ones denote full space. □

3.2.3 Polyhedral complex

3.2.4 Topological product

In this section, we discuss a solid modeling operation highly useful to generate 3D solids from 2D surfaces, as well as 2-complexes from 1-complexes, and to produce their skeletons taking advantage of one or more 0-complexes.

Definition 3.16 (Topological product) A product space is the *Cartesian product* of a family of topological spaces equipped with a natural topology.

The natural topology on a subset of a topological space, in our case $\mathbb{E}^d$, is the relative topology (or subspace topology). The product of two cell complexes can be made into a cell complex as we show in the following.

Definition 3.17 (Product of cell complexes) If X and Y are m- and n-complexes in $\mathbb{E}^m$ and $\mathbb{E}^n$, respectively, $X \times Y \subset \mathbb{E}^{m+n}$ is a cell complex in which each cell is a product of a cell in X and a cell in Y, endowed with the relative topology in $\mathbb{E}^{m+n}$.

3.2.5 Cell complex product

Definition 3.18 (larprod) The `larprod` function takes as input a pair of `Lar` models and returns the model of their *Cartesian product*. Since the `Lar` type is a pair (*geometry, topology*), the output of `larprod( lar1, lar2 )` is the *topological product* of the input `Lar` values.

3.3 Chain complex

3.3.1 Linear chain spaces

3.3.2 Linear chain operators

Matrix-Vector Products as Linear Combinations

Left-multiply a matrix $\boldsymbol{A}$ by a *row vector* $\boldsymbol{x}$, i.e. $\boldsymbol{xA}$, is a linear combination of $\boldsymbol{A}$ rows with $\boldsymbol{x}$ elements as scalars, and produces a row vector. Analogously, right-multiply a matrix $\boldsymbol{A}$ by a *column vector* $\boldsymbol{x}$, i.e. $\boldsymbol{Ax}$, is a linear combination of $\boldsymbol{A}$ columns, and produces a column vector.

In this book we often use a pair of characters from {`V`, `E`, `F`}, for instance `EF`, to denote the mapping `F` $\to$ `E` from 2-chains in `F` to 1-chains in `E` (see Section 3.1).

Coding 3.3.1 (Construction of operator matrix) $[\partial_2]$. With some abuse of language (removed after Section **??**), we build the sparse matrix of `EF` $\equiv$ `∂₂: F → E`.

```
∂₂ = K(EV) * K(FV)' .÷ 2  #=
12×8 SparseMatrixCSC{Int64, Int64} with 24 stored entries:
 ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  1
 ⋅  ⋅  1  ⋅  ⋅  1  ⋅  ⋅
 ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  1
 ⋅  ⋅  1  1  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  1  ⋅  1  ⋅
 1  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅
 ⋅  1  ⋅  ⋅  ⋅  ⋅  1  ⋅
 1  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  1  1  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  1
 1  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  1  ⋅  1  ⋅  ⋅  ⋅  ⋅   =#
```

It may be interesting to make some notation about this matrix. Remember that, by right multiplication, any mapping matrix sends the space of columns (2-chains: triangles here) to the space of rows (1-chains: edges here). With any simplicial complex we have two 1s on each row and three 1s on each column. They characterize the two faces incident on the edge, and the three edges incident on the face, respectively. □

Coding 3.3.2 (How to use the linear operators.) A boundary operator, which is a linear map, sends every vector in domain space (the span of the matrix columns) to its image vector in target space (the span of the matrix rows).

```
face = zeros(length(FV)); face[4] = 1
(∂₂ * face)'  #=
1×12 adjoint(::Vector{Float64}) with eltype Float64:
 0.0  0.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0   =#
```

In particular, the matrix of a linear operator—coordinate representation of it with respect to the chosen bases—send the coordinate representation of a domain vector to its image in target space. Both are binary vectors, as will be clear after Section **??**. E.g., `face[4]` has boundary edges 3, 4, and 12 (look for `1`s positions in row 5 above). □

3.4 Cochain integration (surface, volume, inertia)

Coding 3.4.1 (Volume of a cube from boundary triangles) . The variable `obj` first contains the memory pointer to an object of type `Hpc`, then to an object of type `Lar`. Here, `obj.C` gives the dictionary `C` of its cell complex, where `C[:FV]` denotes the array of boundary faces (triangles).

```
julia> obj = SIMPLEX(3)     #=
Hpc(MatrixNd(4), Geometry([[0.0, 0.0, 0.0], [1.0, 0.0, 0.0],
    [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], hulls=[[1, 2, 3, 4]])) =#


julia> obj = LAR(SIMPLEX(3))    #=
Lar(3, 3, 4, [0.0 1.0 0.0 0.0; 1.0 0.0 0.0 0.0; 0.0 0.0 0.0
    1.0], Dict{Symbol, AbstractArray}(:CV => [[1, 2, 3, 4]], :FV
     => [[1, 2, 3], [2, 3, 4], [1, 3, 4], [1, 2, 4]], :EV =>
    [[2, 3], [1, 3], [1, 2], [3, 4], [2, 4], [1, 4]])) =#

FV = obj.C[:FV]  #=
4-element Vector{Vector{Int64}}:
 [1, 2, 3]
 [1, 2, 4]
 [1, 3, 4]
 [2, 3, 4]   =#
P = (obj.V, FV); VOLUME(P)  #=
0.16666666666666666   =#
volume(P) * 6   #=
1.0   =#
```

References

1. Arakaki., T.: Julia Data Parallel Computing. URL `https://juliafolds.github.io/data-parallelism/tutorials/quick-introduction/`. [retrieved june 27, 2023]
2. Aubanel, E.: Elements of Parallel Computing. Chapman Hall/CRC Press (2016)
3. Backus, J.: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Commun. ACM **21**(8), 613641 (1978). DOI 10.1145/359576.359579. URL `https://doi.org/10.1145/359576.359579`
4. Backus, J., Williams, J., Wimmers, E.: An introduction to the programming language FL. In: D. Turner (ed.) Research Topics in Functional Programming. Addison-Wesley, Reading, MA (1990)
5. Backus, J., Williams, J.H., Wimmers, E.L., Lucas, P., Aiken, A.: FL LANGUAGE MANUAL. PARTS l AND 2. Tech. Rep. RJ 7100 (67163), IBM Almaden Research Center (1989)
6. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review **59**(1), 65–98 (2017). URL `https://doi.org/10.1137/141000671`
7. Carlsson, C.: Syntax highlighting – OhMyREPL. URL `https://kristofferc.github.io/OhMyREPL.jl/latest/`. [retrieved may 22, 2024]
8. Danisc, S., Kavalar, M., Dombrowski, M., Markovics, P.: An Introduction to GPU Programming in Julia. URL `https://nextjournal.com/sdanisch/julia-gpu-programming`. [retrieved june 29, 2023]
9. DiCarlo, A., Milicchio, F., Paoluzzi, A., Shapiro, V.: Chain-based representations for solid and physical modeling. Automation Science and Engineering, IEEE Transactions on **6**(3), 454 –467 (2009). DOI 10.1109/giorgio
10. DiCarlo, A., Paoluzzi, A., Shapiro, V.: Linear algebraic representation for topological structures. Computer-Aided Design **46**, 269–274 (2014). DOI 10.1016/j.cad.2013.08.044. URL `https://doi.org/10.1016/j.cad.2013.08.044`
11. Ferrucci, V.: Generalised extrusion of polyhedra. In: Proceedings on the Second ACM Symposium on Solid Modeling and Applications, SMA '93, p. 3542. Association for Computing Machinery, New York, NY, USA (1993). DOI 10.1145/164360.164376. URL `https://doi.org/10.1145/164360.164376`
12. Ferrucci, V., Paoluzzi, A.: Extrusion and boundary evaluation for multidimensional polyhedra. Comput. Aided Des. **23**(1), 4050 (1991). DOI 10.1016/0010-4485(91)90080-G. URL `https://doi.org/10.1016/0010-4485(91)90080-G`
13. Hatcher, A.: Algebraic topology. Cambridge University Press (2002)
14. Julia: Manual: Base/lib-collections/iteration. URL `https://docs.julialang.org/en/v1/base/collections/#lib-collections-iteration`. [retrieved june 25, 2023]
15. Julia: Manual: Distributed-Computing. URL `https://docs.julialang.org/en/v1/manual/distributed-computing/#Multi-processing-and-Distributed-Computing`. [retrieved june 29, 2023]
16. Julia: Manual: Linear Algebra. URL `https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#man-linalg`. [retrieved july 3, 2023]
17. Julia: Manual: LinearAlgebra.factorize. URL `https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#LinearAlgebra.factorize/`. [retrieved june 25, 2023]
18. Julia: Manual: Metaprogramming. URL `https://docs.julialang.org/en/v1/manual/metaprogramming/`. [retrieved june 21, 2023]
19. Julia: Manual: Modules. URL `https://docs.julialang.org/en/v1/manual/modules/#modules`. [retrieved june 29, 2023]

20. Julia: Manual: Multi-Threading. URL `https://docs.julialang.org/en/v1/manual/multi-threading/#man-multithreading`. [retrieved june 29, 2023]
21. Julia: Manual: Parallel Computing. URL `https://docs.julialang.org/en/v1/manual/parallel-computing/#Parallel-Computing`. [retrieved june 26, 2023]
22. Julia: Manual: Parallel Computing. URL `https://docs.julialang.org/en/v1/base/parallel/#Tasks`. [retrieved june 27, 2023]
23. Julia: Manual: stdlib/SparseArrays. URL `https://docs.julialang.org/en/v1/stdlib/SparseArrays/#man-csc/`. [retrieved june 25, 2023]
24. Julia: Manual: Types. URL `https://docs.julialang.org/en/v1/manual/types/#man-types`. [retrieved june 30, 2023]
25. Julia Community: How to develop a Julia package. URL `https://julialang.org/contribute/developing_package/`. [retrieved june 29, 2023]
26. JuliaGPU: A gentle introduction to parallelization and GPU programming in Julia. URL `https://cuda.juliagpu.org/stable/tutorials/introduction/#Introduction`. [retrieved june 29, 2023]
27. Kamiski, B.: Julia for Data Analysis. Manning (2023)
28. Mainon, P.: Writing type-stable julia code (2021). URL `https://blog.sintef.com/industry-en/writing-type-stable-julia-code/`
29. Paoluzzi, A.: Geometric Programming for Computer Aided Design. John Wiley Sons, Chichester, UK (2003). URL `https://onlinelibrary.wiley.com/doi/book/10.1002/0470013885`
30. Paoluzzi, A., Pascucci, V., Vicentino, M.: Geometric programming: a programming approach to geometric design. ACM Trans. Graph. **14**(3), 266–306 (1995). DOI 10.1145/212332.212349. URL `http://doi.acm.org/10.1145/212332.212349`
31. Paoluzzi, A., Scorzelli, G., Vicentino, M.: Securing the cultural heritage via geometric programming and modeling. Tech. rep., Dept of Computer Science and Engineering, Roma Tre University (2009). URL `https://www.academia.edu/47017676`
32. Paoluzzi, A., Shapiro, V., DiCarlo, A., Furiani, F., Martella, G., Scorzelli, G.: Topological computing of arrangements with (co)chains. ACM Trans. Spatial Algorithms Syst. **7**(1) (2020). DOI 10.1145/3401988. URL `https://doi.org/10.1145/3401988`
33. Paoluzzi, A., Shapiro, V., DiCarlo, A., Scorzelli, G., Onofri, E.: Finite algebras for solid modeling using julias sparse arrays. Computer-Aided Design **155**, 103436 (2023). DOI https://doi.org/10.1016/j.cad.2022.103436. URL `https://www.sciencedirect.com/science/article/pii/S0010448522001695`
34. Rowland, T.: Characteristic function. In: From MathWorld, p. Foundations of Mathematics > Set Theory > Sets. A Wolfram Web Resource (2005). URL `ttps://mathworld.wolfram.com/CharacteristicFunction.html`
35. Scorzelli, G.: Pyplasm library (2023). URL `https://libraries.io/pypi/pyplasm`
36. Scorzelli, G., Paoluzzi, A.: Plasm.jl: v0.1.0 (2023). URL `https://github.com/scrgiorgio/Plasm.jl`
37. Weisstein, E.W.: Julia set. From MathWorld–A Wolfram Web Resource URL `https://mathworld.wolfram.com/JuliaSet.html`
38. Whitaker, S.: Mastering the Julia REPL. URL `https://blog.glcs.io/julia-repl#heading-starting-the-julia-repl`. [retrieved may 22, 2024]
39. Whitney, H.: Geometric Integration Theory. Princeton University Press, Princeton (1957). DOI doi:10.1515/9781400877577. URL `https://doi.org/10.1515/9781400877577`
40. Wikibooks: Introducing julia/dictionaries and sets — wikibooks, the free textbook project (2020). URL `https://en.wikibooks.org/`. [Online; accessed 20-June-2023]

41. Williams, J.H., Wimmers, E.L.: Sacrificing Simplicity for Convenience: Where Do You Draw the Line? In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88, p. 169179. Association for Computing Machinery, New York, NY, USA (1988). DOI 10.1145/73560.73575. URL https://doi.org/10.1145/73560.73575